Index to Constant Weight Codeword Converter

Jon T. Butler† and Tsutomu Sasao‡⋆

†Naval Postgraduate School, Monterey, CA, 93921-5121, USA
`jon_butler@msn.com`
‡Kyushu Institute of Technology, Iizuka, Fukuoka, 820-8502, JAPAN
`sasao@cse.kyutech.ac.jp`

Abstract. A constant weight codeword is a binary n-tuple with exactly r 1's. We show two circuits that generate constant weight codewords. The first is based on the combinatorial number system. Its input is an index to the codeword. That is, there are $\binom{n}{r}$ n-bit codewords with exactly r 1's. The index generates a unique codeword, and is a binary number between 0 and $\binom{n}{r} - 1$. Such a circuit is useful for encoding data. If a *random* constant weight codeword is needed, as in Monte Carlo simulations, then the index is random. If a random constant weight codeword only is needed, then our other circuit is even more compact. It is based on a trellis configuration. Both designs can be pipelined to produce one constant weight codeword per clock period. We give experimental results showing the efficiency of our designs on the SRC-6 reconfigurable computer.

1 Introduction

The generation of an arbitrary n-bit binary word with a fixed weight r (number of 1's) is surprisingly difficult. It is convenient to describe such a word as a *constant weight codeword*, even when discussing a non-coding application.

A constant weight code generator is useful in the enumeration of bent Boolean functions by reconfigurable computer [4, 20]. Bent functions are used in cryptographic applications because they are resilient to a *linear* attack. It is known that the truth tables of n-variable bent functions have one of only two weights, $2^{n-1} \pm 2^{n/2-1}$ [16]. Therefore, instead of testing all 2^{2^n} n-variable truth tables for bentness, it is sufficient to enumerate only functions with these weights. Because the search space of this reduced space is still large, one seeks a Monte Carlo approach, in which random binary numbers, representing truth tables with a fixed number of 1's, are generated.

We have used the constant weight codeword generator in the enumeration of Boolean functions by reconfigurable computer to determine their correlation

Report Documentation Page		

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **MAR 2011**	2. REPORT TYPE	3. DATES COVERED

4. TITLE AND SUBTITLE **Index to Constant Weight Codeword Converter**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,Department of Electrical and Computer Engineering,Monterey,CA,93943**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
A constant weight codeword is a binary n-tuple with exactly r 1's.We show two circuits that generate constant weight codewords. The ?rst is based on the combinatorial number system. Its input is an index to the codeword. That is, there are ?n r ? n-bit codewords with exactly r 1's. The index generates a unique codeword, and is a binary number between 0 and ?n r ? ?1. Such a circuit is useful for encoding data. If a ran- dom constant weight codeword is needed, as in Monte Carlo simulations then the index is random. If a random constant weight codeword only is needed, then our other circuit is even more compact. It is based on a trel- lis con?guration. Both designs can be pipelined to produce one constant weight codeword per clock period. We give experimental results showing the e?ciency of our designs on the SRC-6 recon?gurable computer.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**12**	

immunity [6]. Correlation immunity is a cryptographic property of Boolean functions used in encryption/decryption to determine the extent to which the input values can be determined from the output value. We were able to test one function in each clock period because the index to constant weight code converters shown here can process at one constant weight code per clock period.

Such circuits have application in other areas. For example, Yamanaka, Shimizu, and Shan [23] program the lexicographical generation of constant weight codewords on a reconfigurable computer to analyze energy efficient networks. Balanced codes, in which there are as many 0's as 1's, are useful for encoding data transferred on and off VLSI chips in a way that minimizes the current fluctuation during switching [21]. In other instances of VLSI data transfer, codes with small weight are desired because they yield faster and more compact circuits [21]. Constant weight codes are a countermeasure to "side-channel" attacks [8]. Such attacks use data dependent differences in power consumption to extract secret information. Constant weight codes have been used in asynchronous logic as a means to implement delay-insensitive codes [22]. Three-out-of-six and two-out-of-seven constant weight codes have been used to build a parallel processor for neural simulation [9].

The generation of constant weight codewords in lexicographic order has received much attention spanning 40 years (c.f. [14]). One of the earliest contributions, Gosper's "Hack" [7], is an elegant sequence of basic instructions that can be easily programmed (see also Knuth's MMIX version [10]). The output is a sequence of constant weight codewords whose binary number representation increases each time a codeword is generated. Both normal and reverse lexicographical sequences are useful in heuristics for the generation of sets of codewords with large cardinality [15]. That is, the generation of codewords in either normal or reverse lexicographical order is useful in producing large sets of codewords each a minimum Hamming distance from all other codewords in the set. Interestingly, this bin-packing problem, in which one seeks the *largest* number of constant weight codewords each no less than a specified Hamming distance from another codeword, has remained unsolved for 50 years [1, 13].

Unfortunately, Gosper's Hack is not able to convert an index to a constant weight code corresponding to that index. Thus, it cannot be used in an *encoding algorithm* for constant weight codewords or in a Monte Carlo simulation. In both cases, there is a need to freely choose the sequence of constant weight codewords. A software version of such an encoding algorithm was developed more than 30 years ago [2], but as far as we know, a hardware implementation has never been reported.

We correct this deficiency in this paper. For example, there are $\binom{6}{3} = 20$ 6-bit codewords with exactly three 1's. These can be indexed by a five bit index, whose value ranges from 0 to 19. The index values 20 through 31 are unused. In a coding application, the index will be determined by the plaintext message. In a Monte Carlo simulation, there is an issue related to the index value because typical random number generators produce a value from 0 to $2^n - 1$. We discuss

how to deal with this mismatch. Our architecture is a LUT cascade and is simple. Yet, it produces one constant weight codeword every clock period.

In Section 2, we discuss the combinatorial number system. We show how it can be used to generate constant weight codes, and we discuss its circuit implementation. Then, in Section 3, we discuss a trellis circuit for computing *random* constant weight codewords and compare its complexity/speed with that derived from the combinatorial number system. The trellis circuit is an especially efficient way to generate random constant weight codes. Finally, in Section 4, we give concluding remarks.

2   The Combinatorial Number System

2.1   Introduction

As in the standard binary number system, in the combinatorial number system, each number is represented by a unique vector of basis values.

Definition 1 *In an $\binom{n}{r}$* **combinatorial number system** *[11], integer $N < \binom{n}{r}$ is represented as $N = c_r c_{r-1} \ldots c_1$, where*

$$N = \binom{c_r}{r} + \binom{c_{r-1}}{r-1} + \ldots + \binom{c_1}{1}, \tag{1}$$

such that $c_r > c_{r-1} > \ldots > c_1 \geq 0$.

Example 1 *Table 1 shows the representation of numbers in the $\binom{6}{3}$ combinatorial number system, where $0_{10} \leq N \leq 19_{10}$. The rightmost column of Table 1 shows the 6 bit constant weight code corresponding to N. The constant weight, in this example, is 3, as there are three 1's in each word. Note that the three elements of the vector representation of N correspond to the position of the 1 in the constant weight codeword. For example, 19_{10} or 543 corresponds to 111000, while 0_{10} or 210 corresponds to 000111.*

(End of Example)

We can make the following observations.

1. Regardless of N, each representation has exactly r "digits" (vector elements). For example, a $\binom{7}{3}$ combinatorial number system also has three "digits". However, the basis values are larger than those in a $\binom{6}{3}$ combinatorial number system.
2. Each set of digits, such that $c_3 > c_2 > c_1 \geq 0$, corresponds to a unique value of N. Each value of N corresponds to a *unique* set of digits, such that $c_3 > c_2 > c_1 \geq 0$. It can be seen to be true in this example. This was proven to hold in a general $\binom{n}{r}$ combinatorial number system by Lehmer [12].
3. Given N, a greedy algorithm derives $c_r \ c_{r-1} \ \ldots \ c_1$. For example, in Table 1, the first digit for 19_{10} can be obtained by finding the largest c_3, such that $\binom{c_3}{3} \leq 19_{10}$. Since $\binom{2}{3} = 0_{10}$, $\binom{3}{3} = 1_{10}$, $\binom{4}{3} = 4_{10}$, $\binom{5}{3} = 10_{10}$, $\binom{6}{3} = 20_{10}$, ..., $c_3 = 5_{10}$. Similarly, we seek the largest c_2 such that $\binom{c_2}{2} \leq 19_{10} - 10_{10} = 9_{10}$, etc.

Table 1. The $\binom{6}{3}$ Combinatorial Number System for $0_{10} \leq N \leq 19_{10}$

N	$c_3\ c_2\ c_1$ for $k=3$	Value of N for $k=3$	Constant Weight Codeword
19_{10}	5 4 3	$\binom{5}{3} + \binom{4}{2} + \binom{3}{1} = 10+6+3$	111000
18_{10}	5 4 2	$\binom{5}{3} + \binom{4}{2} + \binom{2}{1} = 10+6+2$	110100
17_{10}	5 4 1	$\binom{5}{3} + \binom{4}{2} + \binom{1}{1} = 10+6+1$	110010
16_{10}	5 4 0	$\binom{5}{3} + \binom{4}{2} + \binom{0}{1} = 10+6+0$	110001
15_{10}	5 3 2	$\binom{5}{3} + \binom{3}{2} + \binom{2}{1} = 10+3+2$	101100
14_{10}	5 3 1	$\binom{5}{3} + \binom{3}{2} + \binom{1}{1} = 10+3+1$	101010
13_{10}	5 3 0	$\binom{5}{3} + \binom{3}{2} + \binom{0}{1} = 10+3+0$	101001
12_{10}	5 2 1	$\binom{5}{3} + \binom{2}{2} + \binom{1}{1} = 10+1+1$	100110
11_{10}	5 2 0	$\binom{5}{3} + \binom{2}{2} + \binom{0}{1} = 10+1+0$	100101
10_{10}	5 1 0	$\binom{5}{3} + \binom{1}{2} + \binom{0}{1} = 10+0+0$	100011
9_{10}	4 3 2	$\binom{4}{3} + \binom{3}{2} + \binom{2}{1} = 4+3+2$	011100
8_{10}	4 3 1	$\binom{4}{3} + \binom{3}{2} + \binom{1}{1} = 4+3+1$	011010
7_{10}	4 3 0	$\binom{4}{3} + \binom{3}{2} + \binom{0}{1} = 4+3+0$	011001
6_{10}	4 2 1	$\binom{4}{3} + \binom{2}{2} + \binom{1}{1} = 4+1+1$	010110
5_{10}	4 2 0	$\binom{4}{3} + \binom{2}{2} + \binom{0}{1} = 4+1+0$	010101
4_{10}	4 1 0	$\binom{4}{3} + \binom{1}{2} + \binom{0}{1} = 4+0+0$	010011
3_{10}	3 2 1	$\binom{3}{3} + \binom{2}{2} + \binom{1}{1} = 1+1+1$	001110
2_{10}	3 2 0	$\binom{3}{3} + \binom{2}{2} + \binom{0}{1} = 1+1+0$	001101
1_{10}	3 1 0	$\binom{3}{3} + \binom{1}{2} + \binom{0}{1} = 1+0+0$	001011
0_{10}	2 1 0	$\binom{2}{3} + \binom{1}{2} + \binom{0}{1} = 0+0+0$	000111

2.2   Circuit Implementation

The keystone of our contribution is a circuit that produces constant weight codewords from an index that is an implementation of an $\binom{n}{r}$ combinatorial number system. Consider, for example, the generation of 6-bit binary words with three 1's. Let $index$ be a 5-bit index whose value specifies which of these codewords is produced. There are $\binom{6}{3} = 20$ codewords, which we assume are specified by $0 \leq index \leq 19$. Let $output$ be a vector of six bits ordered as $output(5)\ output(4)\ output(3)\ output(2)\ output(1)\ output(0)$, three of which are 1 and three of which are 0's. The pseudo-code for this circuit is as follows.

Set $output := 000000$.

```
IF      index ≥ (5 3) = 10,  Set output(5) := 1 and index := index − 10.
ELSEIF  index ≥ (4 3) = 4,   Set output(4) := 1 and index := index − 4.
ELSEIF  index ≥ (3 3) = 1,   Set output(3) := 1 and index := index − 1.
ELSEIF  index ≥ (2 3) = 0,   Set output(2) := 1 and index := index − 0.
```

$$
\begin{array}{lll}
\texttt{IF} & index \geq \binom{4}{2} = 6, & \texttt{Set } output(4) := 1 \texttt{ and } index := index - 6. \\
\texttt{ELSEIF} & index \geq \binom{3}{2} = 3, & \texttt{Set } output(3) := 1 \texttt{ and } index := index - 3. \\
\texttt{ELSEIF} & index \geq \binom{2}{2} = 1, & \texttt{Set } output(2) := 1 \texttt{ and } index := index - 1. \\
\texttt{ELSEIF} & index \geq \binom{1}{2} = 0, & \texttt{Set } output(1) := 1 \texttt{ and } index := index - 0.
\end{array}
$$

$$
\begin{array}{lll}
\texttt{IF} & index \geq \binom{3}{1} = 3, & \texttt{Set } output(3) := 1 \texttt{ and } index := index - 3. \\
\texttt{ELSEIF} & index \geq \binom{2}{1} = 2, & \texttt{Set } output(2) := 1 \texttt{ and } index := index - 2. \\
\texttt{ELSEIF} & index \geq \binom{1}{1} = 1, & \texttt{Set } output(1) := 1 \texttt{ and } index := index - 1. \\
\texttt{ELSEIF} & index \geq \binom{0}{1} = 0, & \texttt{Set } output(0) := 1 \texttt{ and } index := index - 0.
\end{array}
$$

Each of the three IF statements corresponds to the generation of one 1 bit in the codeword, where the first generates the leftmost 1 bit, the second the middle 1 bit and the third the rightmost 1 bit. Fig. 1 shows the circuit that computes the description above. *index* comes in from the left, and *output* exits to the right. Between is a circuit that performs the four operations above. This includes testing the index against a threshold and then performing two operations. The first sets the output to an appropriate value and the second subtracts an appropriate value from the index and passes it on to the next stage, which performs a similar operation. Note that this circuit can be implemented as an r-stage cascade of combinational circuits [17].
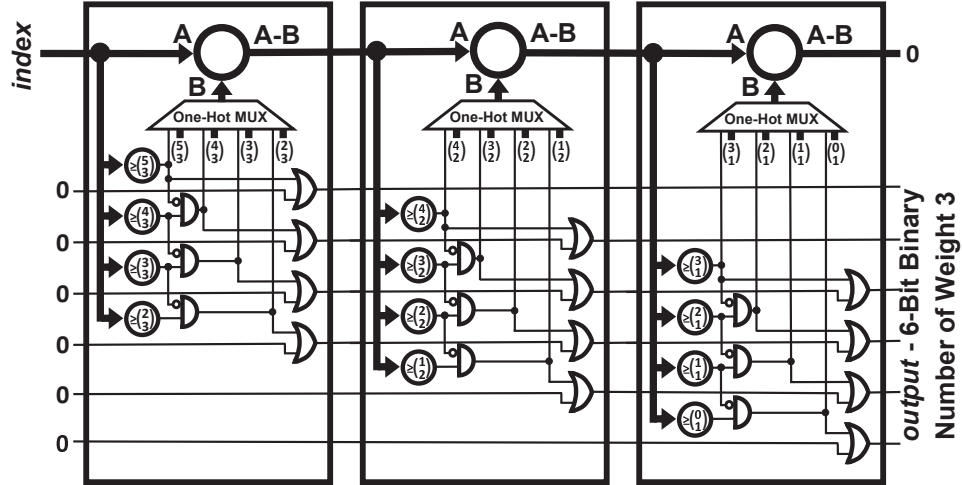

Fig. 1. Example of a Constant Weight Codeword Generator Circuit.

At each stage, there are inputs and outputs that carry a partially completed *output*. Also, there are inputs and outputs that carry *index* reduced by the values contributed by higher order digits. The rightmost stage produces a 0 value at its *index* output, since there are no digits to the right.

Note that this is easily pipelined. Pipeline registers can simply be inserted between stages. Doing so, causes the latency to be r, the weight. Note that, after the first codeword emerges, a codeword emerges at each clock period.

2.3   Results

Fig. 2 shows the result of a program on the SRC-6 reconfigurable computer to produce random constant weight codewords. The SRC-6 uses the Xilinx Virtex2p XC2VP100 FPGA with Package FF1696 and Speed Grade -5. Here, the distribution of constant weight codes with $n = 6$ bits and $r = 3$ 1's is plotted. A total of 1,048,576 $(= 2^{20})$ 64-bit random numbers were generated and converted into random integers from 0 to 19 and applied as indices to the constant weight code generator. For example, the leftmost bar in Fig. 2 corresponds to 52,079 codewords of the form 000111 (=7), and the rightmost bar corresponds to 52,285 codewords of the form 111000 (=56).
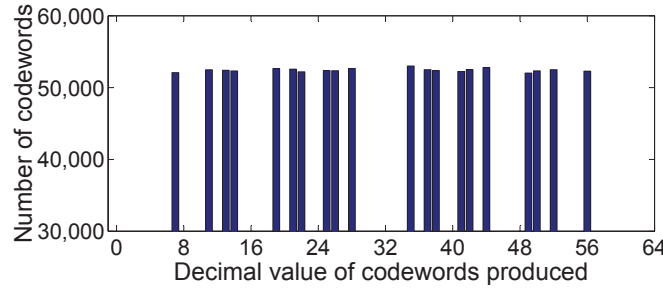
Fig. 2. Distribution of Constant Weight Codewords Produced By the Combinatorial Number System.

The circuit that produced the results in Fig. 2 is shown in Fig. 3. If the *index* is a uniformly distributed random integer over the range $0_{10} \leq N \leq 19_{10}$, then the output is a uniformly distributed set of constant weight codewords. However, if *index* is uniformly distributed over the full range of 5-bit binary numbers, as is common, then 12 *index* values have no corresponding constant weight code. We propose to handle this with a *random number to random integer converter*. In this case, we view the random number generator as producing R, where $0 \leq R < 1$. For example, if the random number generator has 8 bits, then its output is viewed as $0.0000000 \leq R \leq 0.11111111 < 1$. To produce a random integer of value i, where $0 \leq i \leq v - 1$, we form $v \times R$, which can be achieved by integer multiplication of v times the 8 bit integer associated with R following by a division by 256, which corresponds to a right shift by 8 bits. The product vR involves multiplication by a constant. This is done quickly by addition, shift, and truncate. Fig. 3 shows the circuit that realizes this. Note that the random number to random integer converter (i.e. domain converter) consists of all blocks in Fig. 3 except the CWC Generator.

The complete circuit uses 2,880 out of 88,192 4-input LUTs (3%) and 3,804 out of 88,192 slice flip-flops (4%). It can run at 100.1 MHz, which is slightly greater than the SRC-6's operating frequency of 100 MHz. The random number generator is a cellular automata system proposed by Shackleford, Tanaka, Carter, and Snider [19]. As shown in Fig. 3, this is multiplied by v (in this specific example, $v = 20$). The product is right shifted by 64 bits and truncated. The result is a uniformly distributed random integer from 0 to 19. This is applied to
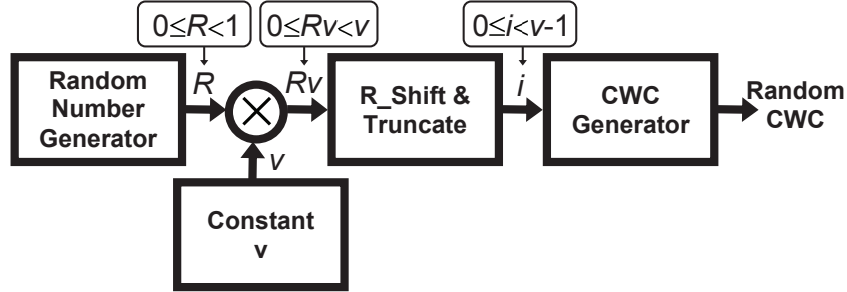
Fig. 3. Block Diagram of a Random Constant Weight Code Generator.

the CWC Generator in Fig. 3, which produces the corresponding constant weight code. A random constant weight codeword is produced at each clock period of the SRC-6's 100 MHz clock. This accounts for 1,048,576 clock periods. However, a total of 1,048,770 clock periods are required. This includes 194 additional clock periods for initialization, data collection, and overhead. Neglecting the overhead, this constant weight codeword generator produces 100 million codewords per second. A C code version of the circuit in Fig. 3 was written and run on the SRC-6's 2.8 GHz Xeon microprocessor. Despite its much higher clock frequency, this implementation produced only 7.6 million random constant weight codewords per second.

2.4   Complexity of Implementation

To understand how the complexity of a $\binom{n}{r}$ combinatorial number system constant weight code generator depends on n and r, we programmed this system for various n and r. Because the SRC-6, with its 130 nm Xilinx Virtex 2p XC2VP100, is a legacy system, we chose the 40 nm Altera Stratix IV EP4SE530F43C3NES FPGA. This is to be used on the SRC Company's newest version of the SRC-7. Table 2 shows the frequency obtained and the resources used in this implementation. A large codeword is achievable (128 bits using 91% of the available ALMs). Although this table shows only balanced constant weight code generators where the number of bits is a power of 2, our approach applies to any number of bits and to any weight.

The second column in Table 2 also shows the number of bits needed to represent `index`. Recall that `index` must be sufficiently large so that all $\binom{n}{r}$ values of the index can be uniquely represented. Specifically, the number of bits needed is $\lfloor log_2 \binom{n}{r} \rfloor + 1$. A naive description in which this was computed as $\lfloor log_2 \frac{n!}{r!(n-r)!} \rfloor + 1$ was not able to give correct results for the bottom half of the table because of the very large value of $n!$. Therefore, this computation was performed as $\lfloor \sum_{i=1}^{n} log_2 i - 2 \sum_{i=1}^{n/2} log_2 i \rfloor + 1$. The Verilog compiler used to implement the circuits lacks the word size to compute $n!$ for moderate n. Also, it is not capable of computing the log_2 function. Instead, a MATLAB program was used. This produced values that were written to a header file that was included in the Verilog code. Similarly, to realize the circuits represented in Table 2, it

Table 2. Frequency and resources used to realize combinatorial number system constant weight code generators on the Altera Stratix IV EP4SE530F43C3NES FPGA.

Con. Wgt. Code $\binom{n}{r}$	#Bits index	Freq. (MHz)	# of LUTs of Various Inputs					Est. # of Packed ALMs	Total # of Registers
			7-	6-	5-	4-	3-		
$\binom{4}{2}$	3	406.3	0	0	0	1	8	5(0%)	10(0%)
$\binom{8}{4}$	7	310.2	0	0	23	30	14	37(0%)	47(0%)
$\binom{16}{8}$	14	213.9	0	40	112	153	66	211(0%)	198(0%)
$\binom{32}{16}$	30	179.7	2	453	719	915	377	1,461(0%)	842(0%)
$\binom{64}{32}$	61	129.5	45	880	983	2,709	44,873	25,428(11%)	3,443(0%)
$\binom{128}{64}$	125	95.8	163	1,422	15,902	14,551	354,448	194,950(91%)	430,608(3%)

was necessary to compute $\binom{n}{r}$, which cannot be realized by 32 bits for moderate values of n and r. Another MATLAB program was written that computed these values and printed them to a header file that was included in the Verilog code.

3   Trellis Generator

3.1   Introduction

An alternative approach for generating random constant weight binary numbers is the trellis circuit. We know of no prior work on this approach except a description in a Japanese book [18] of a program to generate random constant weight codes. A hardware implementation of this is as follows. Fig. 4 shows the trellis circuit of a random constant-weight binary number generator with 6 bits of weight 3. Each bit is generated one at a time starting at top. A complete n-bit number is generated after n clock periods. However, this circuit is pipelined, so that thereafter, a 6-bit binary number of weight 3 is generated at every clock period.

At the top node, the left bit is 0/1 with probability 50%/50%. Depending on this bit's actual value, control goes to the node labeled $\binom{5}{3}$ (0) or the node labeled $\binom{5}{2}$ (1). At this point, a similar process takes place. In this case, the probability of a 0 or 1 bit adjusts so that each n-bit number of weight r has the same probability as any other n-bit number of weight r. In the case of the node labeled $\binom{5}{3}$, 0 and 1 are generated with probability 40% and 60%, while, in the case of the node labeled $\binom{5}{2}$, 0 and 1 are generated with probability 60% and 40%.

The probabilities required for each level are generated by the circuit along the right side of Fig. 4. This circuit generates an integer between 0 and $m-1 > 1$, where m is the level. The top level corresponds to $m = n$, the next lower level to $m = n - 1$, etc.. The nodes in the trellis then convert this number into a 0 or 1, as needed at that node. For example, for $m = 4$, the right-side integer generator produces 0, 1, 2, and 3 with equal probability. For the node labeled $\binom{4}{3}$, a threshold of 1 is used to produce 0 with a probability of 25% (0) and
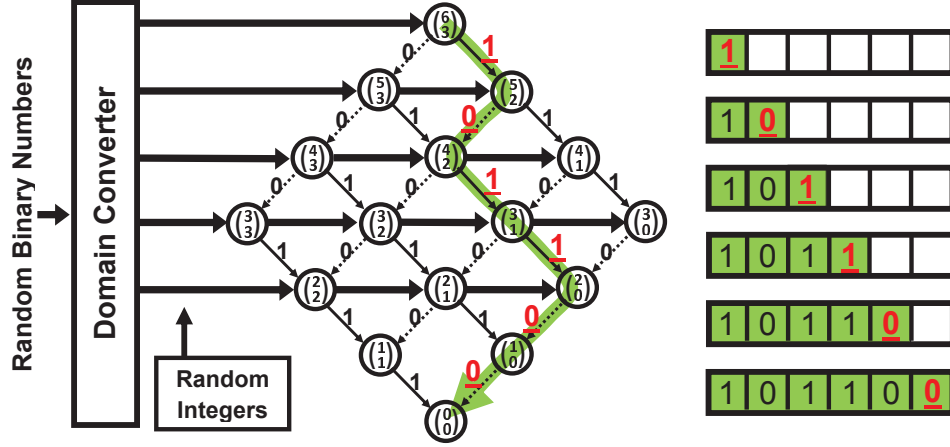
Fig. 4. Trellis Circuit.

1 with a probability of 75% (1, 2, and 3). Similarly, the nodes labeled $\binom{4}{2}$ and $\binom{4}{1}$ use thresholds 2 and 3, respectively. We note that the generation of *one* uniformly distributed integer at level n is sufficient because only one bit of a random constant-weight binary number is generated at that level.

3.2   Circuit Implementation

The trellis can be implemented by a pipeline with n stages. In each stage, the (partially completed) constant weight code is processed and passed to the next stage, as suggested by the words shown along the right side of Fig. 4. Fig. 5 shows the pipeline implementation of the trellis circuit. The block labeled RNG (random number generator) in each stage determines exactly one bit. Whether it is a 0 or 1 is determined at random by a probability that depends on the number of 1 bits generated so far. For example, if all of the r 1 bits occur in the previous stages, then the probability that a 0 is produced by RNG is 100%.

3.3   Results

Fig. 6 shows the distribution of constant weight codes when 1,048,576 ($= 2^{20}$) sets of random numbers are generated and applied to the trellis circuit. This was programmed on the SRC-6. For the Xilinx Virtex2p XC2VP100 FPGA used in this system, the complete circuit uses 2,767 out of 88,192 4-input LUTs (3%) and 3,795 out of 88,192 slice flip-flops (4%). It can run at 106.2 MHz, which accommodates the SRC-6's fixed 100 MHz clock. It produces one constant weight codeword at each 100 MHz clock cycle, and takes a total of 1,048,766 clock cycles. This includes 90 clock periods in addition to those 1,048,576 clock cycles that each produce a constant weight codeword. Each level in the trellis uses a 16 bit random number generator that creates a random integer generator, as described above. Note that the resources just described include circuits needed to
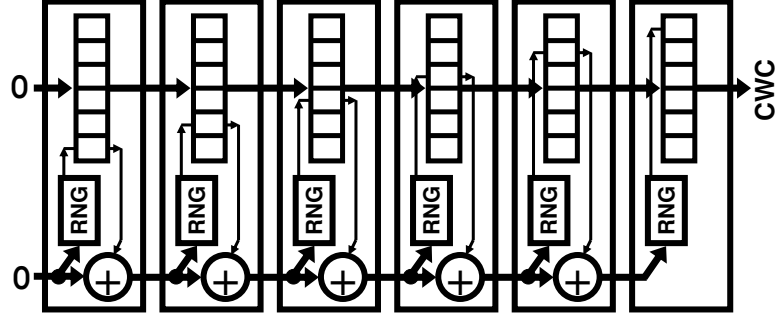
Fig. 5. Block Diagram of the Trellis Constant Weight Code Generator.

implement overhead functions like data collection. Neglecting the overhead, the trellis random constant weight codeword generator produces 100 million codewords per second. A C code version of the circuit in Fig. 5 was written and run on the SRC-6's 2.8 GHz Xeon microprocessor. Despite its much higher clock frequency, this implementation produced only 57.2 million random constant weight codewords per second.
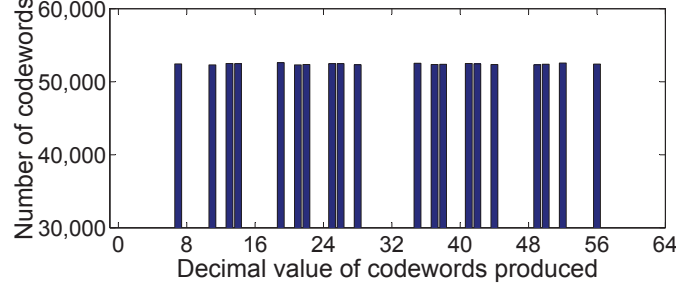
Fig. 6. Distribution of Constant Weight Codewords Produced by the Trellis Circuit.

3.4   Complexity of Implementation

Like the combinatorial number system constant weight codeword generator, the trellis was programmed on the Altera Stratix IV EP4SE530F43C3NES FPGA. Table 3 shows the resource usage and frequency for various types of constant weight codewords. Unlike the combinatorial number system constant weight codeword generator, 256-bit constant weight codewords with 128 1's can be easily implemented within one FPGA. The Estimated Number of ALMs increases by a factor of more than 2 for each one line advance in Table 3. A similar statement is approximately true of the column labeled Total Number of Registers. The resources shown in Table 3 do not cover the circuits that perform the overhead functions, such as data collection; they include only the trellis circuit.

Table 3. Frequency and resources used to realize a trellis constant weight code generator on the Altera Stratix IV EP4SE530F43C3NES FPGA.

Con. Wgt. Code $\binom{n}{r}$	Freq. (MHz)	# of LUTs of Various Inputs					Est. # of Packed ALMs	Total # of Registers
		7-	6-	5-	4-	3-		
$\binom{4}{2}$	487.6	0	0	1	24	83	42 (0%)	65 (0%)
$\binom{8}{4}$	463.8	0	0	19	12	71	122 (0%)	172 (0%)
$\binom{16}{8}$	344.4	0	6	28	33	281	358 (0%)	444 (0%)
$\binom{32}{16}$	274.2	0	31	96	101	765	978 (0%)	1,137 (0%)
$\binom{64}{32}$	250.2	0	71	289	365	1,994	2,839 (1%)	3,387 (0%)
$\binom{128}{64}$	231.1	1	301	908	941	4,673	5,344 (2%)	4,691 (1%)
$\binom{256}{128}$	174.9	1	2,757	2,363	10,653	4,673	11,309 (5%)	8,011 (1%)

4   Concluding Remarks

Although there is a need for a circuit that computes constant weight codewords from indices, we have not seen an implementation. Our results are useful, for example, in the encoding/decoding of data, such as between on-chip and off-chip and in asynchronous circuits. We show two approaches 1) a combinatorial number system implementation and a 2) trellis implementation. The combinatorial number system can be implemented as a pipeline producing a constant weight codeword at each clock. For a constant weight code of n bits, of which r are 1, the pipeline is r stages long. If one only wants to produce random constant weight codewords, for example in Monte Carlo simulations, the trellis is also implemented efficiently. Its pipeline is n stages long. The trellis requires less resources and operates at a higher frequency. We have implemented both designs on the SRC-6 reconfigurable computer and on an Altera Stratix IV EP4SE530F43C3NES FPGA. This has shown that both are efficiently implemented.

We remark on two extensions of our results. First, both circuits can be implemented as combinational logic. Indeed, the diagrams, Figs. 1 and 5, show them as combinational logic. Second, in both circuits, the probability of certain codewords can be controlled.

References

1. A. E. Brouwer, J. B. Shearer, N. J. A. Sloane, and W. D. Smith, "A new table of constant weight codes", *IEEE Trans. Infor. Theory*, Vol. 36, No. 6, pp. 1134–1380,(1990)
2. B. P. Buckles and M. Lybanon, "Algorithm 515: Generation of a vector from the lexicographical index [G6]", *ACM Transactions on Mathematical Software*, Vol. 3, Issue 2, pp. 180–182, (1977)

3. G. Bubniak, M. Goralczyk, M. Karp, and Z. Kokosinski, "A hardware implementation of a generator of (n,k)-combinations," *Proc. IFAC Workshop "Programmable Digital Systems" PDS'2004*, Krakow pp. 228–231, (2004)

4. J. T. Butler and T. Sasao, "Boolean functions for cryptography," in *Progress in Applications of Boolean Functions*, Morgan & Claypool Publishers, pp. 33-54, (2010)

5. Combinadic - http://en.wikipedia.org/wiki/Combinadic .

6. C. Etherington, "An analysis of cryptographically significant functions with high correlation immunity by reconfigurable computer," M.S. Thesis, December (2010)

7. Gosper, R. W. Item 175 in Beeler, M., Gosper, R. W., and Schroeppel, R., "HAK-MEM," Cambridge, MA: MIT Artificial Intelligence Laboratory, Memo AIM-239, Feb. (1972). http://www.inwap.com/pdp10/hbaker/hakmem/hacks.html#item175.

8. http://en.wikipedia.org/wiki/Side_channel_attack .

9. M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras, and S. B. Furber, "SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor," *Inter. Joint Conf. on Neural Networks (IJCNN)*, pp. 2850–2857, (2008)

10. D. E. Knuth, *The Art of Computer Programming*, "Bitwise tricks and techniques," Vol. 4, **Fascicle 1**, Addison-Wesley, p. 152, ISBN 0-321-58050-8.

11. D. E. Knuth, *The Art of Computer Programming*, "Generating all combinations and partitions," Vol. 4, **Fascicle 3**, Addison-Wesley, pp. 5-6, ISBN 0-321-58050-8, (2009).

12. D. H. Lehmer, "The machine tools of combinatorics," in *Applied Combinatorial Mathematics*, E. F. Beckenbach, Ed., Wiley, New York, pp. 5–30, (1964)

13. F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*, Amsterdam, North-Holland, (1977)

14. A. Nijenhuis and H. S. Wilf, , *Combinatorial Algorithms*, 2nd Edition, Academic Press, (1978)

15. R. Montemanni and D. H. Smith, "Heuristic algorithms for contructing binary constant weight codes," *IEEE Trans. on Inform. Theory* Vol. 55, No. 10, pp. 4651–4656, (2010)

16. O. S. Rothaus, "On 'bent' functions", *J. Combinatorial Theory*, Ser. A, Vol. 20 pp. 300–305, (1976),

17. T. Sasao, *Memory Based Logic Synthesis*, 1st Edition, Springer, ISBN 978-1-4419-8103-5, (2011)

18. I. Senba, *Combinatorial Algorithms*, (in Japanese), Science Inc., Tokyo, (1989)

19. B. Shackleford, M. Tanaka, T. Carter, and G. Snider, "High-performance cellular automata random number generators for embedded probabilistic computing systems," *Proc. of 2002 NASA/DOD Conf. on Evolvable Hardware*, IEEE Computer Society, pp. 191-200, July (2002).

20. J. L. Shafer, S. W. Schneider, J. T. Butler, and P. Stanica, "Enumeration of bent Boolean functions by reconfigurable computer," *The 18th Annual Inter. IEEE Symp. on Field-Programmable Custom Comput. Mach. (FCCM-2010)*, Charlotte, NC, May 2-4, pp. 265–272, (2010).

21. L. G. Tallini and B. Bose, "Design of balanced and constant weight codes for VLSI systems," *IEEE Trans. on Computers*, Vol. 47, No. 5, pp. 556–572, (1998)

22. T. Verhoeff, "Delay-insensitive codes - an overview," *Distr. Comp*, 3(1):1-8, (1988)

23. N. Yamanaka, S. Shimizu, and G. Shan, "Energy efficient network design tool for green IP/ethernet networks," *ONDM2010*, Jan. 31 - Feb. 3, 2010, Kyoto, Japan, (2010)